LISP Theory & Practice

ACORNSOFT

Acknowledgements

This book is based on contributions by Mike Gardiner and Charles G Smith. The Publishers would also like to thank Dr Arthur Norman for valuable comments on the draft copy.

FIRST EDITION

ISBN 0 907876 01 3

Published by:

Acornsoft Limited
4a Market Hill
Cambridge
CB2 3NJ
England

LISP Theory and Practice

CONTENTS

1 About LISP

LISP was invented in the 1960's by the American mathematician and computer scientist John McCarthy, and it has since become one of the world's major languages for Artificial Intelligence research. It has also proved to be a valuable tool in fields as diverse as symbolic logic and medicine.

If you wish to compute payrolls or construct vast mathematical models of the world's weather systems then LISP is not for you. These applications conform to the conventional view of computing, that of ultra-fast arithmetic performed over and over again on fixed-format data. LISP can do arithmetic but its real strength lies in the ease with which it can handle flexible structures containing data which may or may not have anything to do with mathematics. Let us take our comparison a step further with the following summary:

Conventional Language

Applications:
Finance, Engineering, Applied Maths, Physics.

Data Structures:
Rigid rectangular arrays.

Data Processing:
Fast loops, high precision, floating-point 'number-crunching'.

LISP

Applications:
Artificial Intelligence, Life-sciences, Database Management, Pure Maths, 'intelligent' computer games.

Data Structures:
Flexible structures which may be grown or pruned as necessary, e.g. lists, trees and graphs.

Data Processing:
List processing, symbol manipulation, recursive functions.

The concept of list processing is fundamental to LISP and in fact the name of the language is a contraction of this term. As you learn LISP you will discover just how powerful a programming tool list processing can be - in fact you may wonder how you ever managed without it!

Using this Manual

Chapter 3 gives a practical introduction to programming in LISP. After working through this, the beginner should be able to follow the sections on using ATOM LISP, and the descriptions of additional functions and atoms in the Glossary.

Those who already know LISP should skip the introduction to programming. Chapter 2 'Using ATOM LISP', and Chapter 4, 'Implementation' give the main details of this implementation, and the rest of the manual and Glossary can be referred to as and when necessary.

2 Using ATOM LISP

2.1 Loading LISP

ATOM LISP is supplied on cassette and is designed to work on a 8K + 12K ATOM. To load the system, check the cassette is rewound to the start, place it in the cassette recorder, and enter:

```
*RUN"LISP"
```

followed by RETURN. The ATOM will respond with the message:

```
PLAY TAPE
```

whereupon you should set the cassette recorder to play, and then press RETURN. 'Snow' will appear on the screen as the LISP interpreter is loaded into the graphics memory from #8200. The 3K of LISP utilites and constants supplied on the cassette are then automatically loaded into memory starting at #2800. The system starts by displaying the copyright message, followed by the prompt:

```
^EVALUATE:
```

LISP is now ready to accept your programs.

To leave LISP, press the BREAK key. The system can be re-entered without destroying the user's objects or data structures by typing:

```
LINK#8200
```

followed by RETURN.

2.2 Saving Sessions

The user can save the whole of the LISP memory area currently in use (the 'image') on tape. This includes system functions and data-structures as well as those which have been defined by the user during the course of the session. The image can be reloaded and the session continued from the same point. The commands are:

```
(SAVE '<filename>)
(LOAD '<filename>)
```

for example:

```
(SAVE 'IMAGE1)
```

The LOAD command can be used only when LISP is already running. If you wish to perform a 'cold' start you can use the following sequence:

```
End of Session 1.
Enter:

(SAVE 'IMAGE1)
```

IMAGE is the name chosen for the file in which the memory image will be saved.

Start of Session 2
Enter:

*RUN "LISP"

followed by:

(LOAD 'IMAGE1)

2.3 DOS and COS commands

All the usual commands to the ATOM Disc or Cassette Operating system can be accessed. This is done from within LISP by using a function which has the name '*', thus:

(* '<command>)

There must be at least one space between '*' and the command. Notice also the quote mark in front of the command name. For example:

^EVALUATE: (* 'CAT)

This will give a catalogue of the files on disc or cassette. Abbreviations of DOS commands are not allowed.

3 Programming in LISP

3.1 Introduction

In this chapter of the manual you will learn LISP the way you learned your native language - by using it. We shall be following a practical approach using examples. Each one will teach you some new facts about LISP and as a bonus you will at the same time be introduced to some useful programming techniques.

Every computer has a set of rules telling you what you can and cannot write. The technical term for such a set of rules is the syntax of the language. The syntax of LISP is simpler than that of BASIC, FORTRAN, Algol, COBOL and probably any other commonly used language. This has several advantages, but there is a price to pay: statements which look quite different in BASIC, for example, will appear superficially the same when written in LISP, and this makes them more difficult to read when you are first learning.

Let's do a brief comparison:

BASIC	LISP
3 + 7	(PLUS 3 7)
LET X = 1	(SETQ X 1)
PRINT X	(PRINT X)
"A B C"	(QUOTE (A B C))
PRINT "A B C"	(PRINT (QUOTE (A B C)))
X > 5	(GREATERP X 5)
IF X=1 THEN LET Y=2	(COND ((EQ X 1) (SETQ Y 2)))

In each case the LISP function consists of a series of items in brackets; The first item is always a function name, any remaining items are known as the arguments of the function - the function acts on them to produce its result. You will notice that LISP has rather odd choices for some of its function names. These will be dealt with later. Most people find all the left and right brackets confusing to begin with but, don't worry, your eye will quickly become practised at sorting them out.

3.2 Arithmetic

Load LISP as described in Section 2.1. You should see:

```
^EVALUATE:

Enter

(PLUS 2 2)
```

followed by RETURN. LISP will reply as follows:

```
^VALUE IS: 4
```

Now try adding the following two numbers but leave off the final parenthesis:

```
^EVALUATE: (PLUS 7 8            <RETURN>
```

This time LISP answers by printing a single left-pointing arrow. This is its way of reminding you that it expects brackets to match up in pairs. The system will not evaluate the expression until you have completed it by typing in the missing bracket. To make an expression clearer, you can type RETURN at any point where a space is allowed.

Enter a single right-parenthesis. LISP will now evaluate the expression and print the answer:

```
^VALUE IS: 15
```

Look at the following expressions and decide what answer LISP will give to them. Try each one on the ATOM to see if you were right.

a) (PLUS 1 2 3 4)

b) (PLUS 17)

c) (PLUS)

d) (PLUS (PLUS 2 2) (PLUS 2 2))

e) (PLUS 5 (PLUS 3 4) 2)

Check that you understand the PLUS function by making up some expressions of your own. If you have problems refer to the definition of PLUS in the Glossary.

ATOM LISP has five basic arithmetic functions:

PLUS	addition
DIFFERENCE	subtraction
TIMES	multiplication
QUOTIENT	integer part of division
REMAINDER	remainder of division

They are all used in the same way as PLUS, except that TIMES is the only other one that can take more than two arguments. Try out some expressions using them. How would you translate the following into LISP ?

```
2 * 4  +  3 * 5  +  7
```

The answer is 30 - try out your expression to see if it gets it right.

3.3 Lists

We've seen that LISP can do arithmetic but now let's get on to what the language is really about - list processing. First of all though we have to know what a list looks like. Here are some examples:

a) (ORANGE APPLE PEAR)

b) (76 43 21 82)

c) (VALUES 7 23 42)

d) ((A B) (C D) (E F) (G H))

Note the similarity between these lists and the arithmetic expressions we gave LISP. In fact if VALUES was the name of a function, example (c) could be a valid expression. This illustrates the most important feature of LISP - program and data are represented and stored in exactly the same way. The term s-expression (short for symbolic expression) is used to collectively describe lists and 'atoms', the other major LISP object, when we are not concerned whether they are program or data.

The separate items inside the brackets of a list are called the elements of the list. In example (d) the elements are themselves lists. In the other examples they are LISP atoms, so called because, unlike lists, they cannot be further divided. These atoms are of two kinds: number atoms and character atoms. Number atoms are integer numbers for use in arithmetic calculations, and character atoms are used as descriptive data. PLUS is an example of another atom type: is it a Subr atom (meaning subroutine atom) and it references the machine-code routine that performs its function.

To tell LISP when a list is to be interpreted as data, rather than program for evaluation, the quote operator is used. A single quote before a list or atom turns off evaluation for that item, and LISP leaves it unaltered for use as data. If you enter any one of the above example lists, LISP will give an error because they are not legal programs. A preceeding quote prevents this, for example:

```
^EVALUATE: '(ORANGE APPLE PEAR)

^VALUE IS: (ORANGE APPLE PEAR)
```

LISP created an internal representation of the list, and in the absence of any function using it, has recreated the description from the structure. Try entering the other lists as data in this way. Put extra spaces between the items and note that these are removed in LISP's printed version. Generally speaking, spaces (and carriage returns) can be inserted anywhere between items without upsetting LISP. Also try entering some character atoms as data, noting that normal character atoms cannot contain spaces. LISP will return them as it did lists.

3.4 List Processing

List processing consists of picking out items from existing lists and building up new ones from those items. It is a matter of constructing and rearranging data structures. Surprisingly it turns out that almost all the operations we could ever want to perform on lists can be done using just three simple functions. They are:

CAR, CDR and CONS

CONS is quite sensibly named in that it is used for constructing lists. The functions of CAR and CDR just have to be memorised.

Enter the following:

```
(CAR '(A B C))
```

not forgetting the quote mark. When you press RETURN you should get the reply:

```
VALUE IS: A
```

Now try the following examples:

a) (CAR '(P Q R))

b) (CAR '(A))

c) (CAR '((A B) (C D)))

d) (CAR '(PLUS X Y))

Do you see what the function CAR does? It extracts the first element of the list given as its argument. Next try this:

^EVALUATE: (CDR '(A B C))

You should get:

VALUE IS: (B C)

Can you predict what effect CDR will have on each of the examples above? Try it on each of them. It should be clear that CDR takes a list and returns it with the first element removed.

Enter:

(SETQ L '(A B C D))

LISP will reply with:

^VALUE IS: (A B C D)

SETQ is the equivalent of the BASIC assignment operator '=' : it sets the value of a variable to the result of an expression. In this case L is the variable, properly called the 'data object', and (A B C D) is its new value. The quote has the same effect as before. Try evaluating L. This should be the result:

^EVALUATE: L

^VALUE IS: (A B C D)

We can now use L instead of (A B C D) in expressions.

Enter the following:

(CDR L)

What did you get? What would the answer be if you entered:

(CAR (CDR L))

Try various combinations of CAR and CDR on the list L. See if you can pick out each individual atom. If you get error messages, look them up in Chapter 8 and try to find out where you went wrong.

An important point to note is that when working on a list of atoms, CAR always produces an atom and CDR always produces a list. The exception to this last rule occurs if you remove all the elements from a list with CDR, for example:

^EVALUATE: (CDR (CDR (CDR (CDR L))))

^VALUE IS: NIL

NIL is a special atom that is used to mark the end of lists. It is equivalent to the 'empty list', the list with no elements:

```
^EVALUATE: ()

^VALUE IS: NIL
```

The CONS function constructs lists from atoms and lists. In a sense, it is the opposite of CAR and CDR because if you take the CAR and CDR of L you can CONS them back together to make L again, i.e.

```
^EVALUATE: (CONS (CAR L) (CDR L))

^VALUE IS: (A B C D)
```

Try the following examples using CONS.

a) `(CONS 'A '(B C D))`

b) `(CONS '(A) '(B C D))`

c) `(CONS '(A B) '(C D))`

d) `(CONS '(A B) '((C D)))`

e)
```
(SETQ L '(A B))
(SETQ M '(C D))
(CONS (CAR L) M)
```

f) `(CONS (CAR L) (CDR M))`

g) `(CONS (CAR L) (CDR L))`

CONS is then the function which adds items on to the front of an existing list.

We made the claim earlier that CAR, CDR and CONS could cope with most problems of list processing. You might be wondering how this can be. How, for instance, would you use them to extract the last item from a list containing a thousand items? In the next few pages we shall demonstrate how simply LISP can cope with problems such as this and, in the process, you will discover just how powerful a language it can be.

One last function we need before we can start writing full programs is COND, a conditional function. This is LISP's equivalent of the BASIC IF statement. Here's a comparision:

BASIC	LISP
`IF X=2 THEN Y=Y+1`	`(COND ((EQ X 2) (SETQ Y (PLUS Y 1)))`

COND takes a variable number of arguments of the form:

`(<condition> <result>)`

The 'condition' expression is first evaluated. If the result is NIL, evaluation carries on with the next expression pair. If the result is not NIL, the 'result' expression is evaluated. The value returned by COND is NIL if all of the conditions returned NIL, or the value of the the result expression that was evaluated. More than one result expression can be included after a condition expression: they are evaluated in order and the value returned is that of the last one.

3.5 Functions

Enter the following:

```
(OBLIST)
```

This function provides you with the object list, a list of all the data objects, including functions, which have so far been defined. Have a look through the list and you should recognise the ones that we have used already. To prevent the list from scrolling off the screen before you have read it, type ESC when it has reached the required point (ignoring the resulting error message).

Let us invent a new function and place it on the object list. Suppose we needed a function which would accept any number you gave it and return that number with one added to it. Thus, if we named the new function ADD-ONE, we would like it to work as follows:

```
EVALUATE:  (ADD-ONE  23)
VALUE IS:  24
```

First of all we will write a specification for the function in plain English and then see how this translates into LISP.

Specification of ADD-ONE in English:

'Define a new function named ADD-ONE that will act on any number that we give it (let us call it X) to give the result X plus 1'.

Translation:

English	LISP
Define a new function	(DEFUN...)
named ADD-ONE	(DEFUN ADD-ONE...)
that will act on any number X	(DEFUN ADD-ONE (X)...)
to give the result X plus 1	(DEFUN ADD-ONE (X) (PLUS X 1))

Type in the definition of ADD-ONE exactly as given in the last line above. Now look at the object list - your new function name should be right at the beginning of it. Try the function out:

a) `(ADD-ONE 3)`

b)
```
(SETQ A 94)
(ADD-ONE A)
```

c) `(ADD-ONE (PLUS A A) )`

Have a look at the way the definition has been stored by entering its name:

```
^EVALUATE: ADD-ONE
```

You will notice that it appears slightly different - beginning with the word 'LAMBDA' - we will deal with this shortly. Meanwhile try entering

```
(SPRINT ADD-ONE)
```

The function SPRINT prints out LISP expressions in a format that makes them easier to read and understand.

DEFUN is shorthand for SETQ ... LAMBDA. We have seen that SETQ allows you to give an atom a value and that the value may be a number, a list or the name of another atom. What we haven't mentioned so far is that the value may also be a function defintion. To warn the LISP interpreter to expect a function we use the flag word LAMBDA. Here's an an example of the use of a LAMBDA definition:

```
((LAMBDA (X) (PLUS X 1)) 3)
```

and here is what it means:

LISP	English
((LAMBDA (X) ...	Apply an anonymous function to an argument (call it X)
... (PLUS X 1)) ...	the definition of the function is 'add one to the argument'
... 3)	the argument in this case is 3

This of course is just a long-winded way of saying:

```
(PLUS 3 1)
```

In a realistic case, the function body would be unlikely to be as simple as (PLUS X): it is more likely to extend to many lines and possibly use X more than once. Bearing that in mind, we can see the usefulness of LAMBDA by using it in conjuction with SETQ, thus:

```
(SETQ ADD-ONE '(LAMBDA (X) PLUS X 1)))
```

This creates the data object called ADD-ONE and assigns to it the LAMBDA expression that follows. At this stage the LAMBDA expression is treated as an ordinary list. The difference comes when we use ADD-ONE as though it were a function, i.e. as the first element of list for evaluation. For example:

```
^EVALUATE: (ADD-ONE 7 8)
```

When LISP tries to evaluate this expression, it looks at the first item expecting it to be a function. In fact, what it finds is the name of a data object (ADD-ONE). The system does not give up yet however - it will evaluate anything occupying the function position of an expression up to twice in an attempt to find a valid function. What it finds when it evaluates ADD-ONE is a LAMBDA expression and, as we said, LAMBDA expressions are perfectly good functions in LISP.

At this point, the computer has converted

```
(ADD-ONE 3)
```

into

```
((LAMBDA (X) (PLUS X 1)) 3)
```

You should recognise this! It is the expression which we dissected at the beginning of this discussion.

To summarise: ADD-ONE is a data object and not a function. It (or

any other data object we choose) can however serve as a very useful abbreviation for a function definition in the form of a LAMBDA expression.

This combination SETQ ... LAMBDA is used so frequently that it itself has an abbreviation, this of course is our friend DEFUN.

Have a look at the following function definitions. What do the functions do?

a) (DEFUN P2 (X) (TIMES X X))

b) (DEFUN P4 (X) (P2 (P2 X)))

Enter these definitions and then try the functions out. What happens if you use large numbers as their arguments? Look up the error message you get in Chapter 8. Notice that function P4 makes use of function P2 so you must have typed that one in first.

Think up some arithmetic functions of your own and enter their definitions. Do they work exactly as you expected?

Functions can be made to accept several arguments by including the appropriate number of identifiers in the list after the function name, like the following:

(DEFUN MYFUNCTION (X Y Z) <body of function definition>)

MYFUNCTION will take three arguments which can then be used in the body under the names X, Y, and Z.

Now let us attack the problem of producing the last item of a list of arbitrary length. This is elegantly solved using a function. The reasoning that follows may seem a little strange at first and you may need to read the next few paragraphs more than once. Nevertheless if you can understand them you will have discovered the essence of programming in LISP. Check the following statements:

Imagine that the function we want already exists and is called LAST, then:

a) If there were only one item in the list (i.e. if the CDR of the list is NIL) the answer is the first element of that list. For example:

   (LAST '(P)) should evaluate to P

b) If the list had more than one item then, if you were to take the CDR of it, its last item would be the same as the last item of the list itself. For example:

   (LAST '(P Q R S)) evaluates to S

   and

   (LAST '(Q R S)) also evaluates to S

   Another way of saying this is:

   (LAST '(P Q R S)) is the same as (LAST (CDR '(P Q R S))).

When you have untangled what the above actually means, it probably does not look particularly world-shattering, yet amazingly it contains all the information we need for our definition of the function LAST!

Let us write it out again more concisely:

If X is a list of only one element:

```
Then       (LAST X) evaluates to (CAR X)
Otherwise  (LAST X) evaluates to the same as (LAST (CDR X))
```

This looks remarkably like a definition. Let us try defining the function:

English	LISP
Define a function called LAST which takes a list	(DEFUN LAST (X) ...
And returns a value which, if:	... (COND ...
a) the list contains only one item	... ((EQ NIL (CDR X)) ...
is the first element of the list removed,	... (CAR X))
b) else,	... (T ...
is the result of applying LAST to the list with the first element removed.	... (LAST (CDR X))))

Some of this needs further explanation but first of all let's assemble the bits to make the complete definition:

```
(DEFUN LAST (X)
   (COND
      ((EQ NIL (CDR X)) (CAR X))
      ( T (LAST (CDR X))) ))
```

In case you're not convinced that this will work let us type it in straight away and try it.

Type in the definition of LAST exactly as given. Make sure that you get all the left and right brackets in the proper places. Notice that the two right brackets on their own at the very end are the ones which match up with those in front of DEFUN and COND at the beginning of the function. Notice also that LISP lets you spread a definition over several lines by keeping count of the brackets. Now try out the new function:

```
a)  (LAST '(R S T))
b)  (LAST '(Z))
c)  (LAST '((JIM ALISON) (ARNOLD MARY)))
d)  (LAST 56)
e)  (LAST 'A)
f)  (LAST ())
```

The first three cases should work and the last three should fail. Why?

Let us take a little time out here to make sure you understand the definition of LAST. It should be clear that

```
(EQ NIL (CDR X))
```

detects lists containing only one item because we have already seen that the end of a list is marked by a NIL in the CDR position, and EQ returns T (the truth value) if its arguments are equal.

The second part of the COND is:

```
(T (LAST (CDR X)))
```

The condition is just T so this result expression will always be evaluated if none of the preceding conditions turned out to be true.

(LAST (CDR X)) is the really interesting part of the LAST function and the one that does all the work. It is an example of a recursive function call.

The extreme in recursive definition might be something like this:

```
(DEFUN LAST (X)
       (LAST  X))
```

LISP would accept this as a definition but when trying to evaluate an application of the function would just go round in circles, always expecting the answer to appear next time around. However, it would not go on forever, as would an infinite loop. This is because at each call of LAST, the system creates a new 'reincarnation' of the function that is hoping to receive an answer from its successor which it may pass back to its predecessor. If the level of recursion gets too deep, the machine will eventually run out of space for storing all the versions of the function.

If you would like to see just how far LISP will go before running out of space, try the following:

```
^EVALUATE: (DEFUN FOREVER (N)
<(PRINT 'INCARNATION BLANK N)
<(FOREVER (PLUS N 1)))
^VALUE IS: (LAMBDA (N) (PRINT (QUOTE
INCARNATION) BLANK N) (FOREVER (PLUS N 1)))

^EVALUATE: (FOREVER 1)
```

If you wish to investigate along this line further, you may like to experiment with putting PRINT statements into the following definition of Ackermann's Function:

```
(DEFUN ACK
   (M N)
   (COND
      ((ZEROP M) (PLUS N 1))
      ((ZEROP N) (ACK (DIFFERENCE M 1) 1))
      (T (ACK (DIFFERENCE M 1) (ACK M (DIFFERENCE N 1))))))
```

Ackermann's function was originally devised to prove an abstruse point about the mathematical theory of recursive functions. It is of theoretical interest because it cannot be defined without recursion. The mathematical definition of Ackermann's Function is as follows:

```
Ack [m,n]  =  n+1                          if m=0
              Ack [m-1,n]                  if n=0
              Ack [m-1, Ack [m,n-1]]       otherwise
```

Since it is doubly recursive you will find that its arguments should be kept very small if LISP is not going to run out of space, for example:

```
^EVALUATE: (ACK 2 3)
```

We've seen the dangers of circular definitions which cause infinite recursion. Why then, can we rely on our original version of LAST working? The answer is that at each incarnation of the function, the problem has been reduced slightly beause the CDR of a list will always be shorter than the list itself. Eventually a list containing only one member will be reached and the incarnation presently operating will detect it. It will then hand its answer (which in this case is the answer to the whole problem) back to its predecessor which will in turn hand it on until it reaches the very first call of the function. At this point it is handed up one more level to the LISP system which will print the final answer.

Writing recursive functions is a knack which comes with practice - here are some hints to get you started.

1 Read existing function definitions and try to understand them thoroughly, (see Chapter 7). Start with simple examples.

2 Experiment with minor modifications to existing functions. Try inserting PRINT statements at strategic points.

3 When designing your own recursive functions from scratch, always start by imagining that the required function already exists and then use it (but on a smaller argument) within your definition.

4 Make sure that your end conditions (e.g. detecting empty lists) cover every eventuality.

5 Work through your function by hand as though you were the computer.

6 When trying out the function on the computer, start by testing all the end conditions. Observe that very often you can use very simple cases, e.g. NIL, as end conditions: many recursive LISP functions look like this:

```
(DEFUN XX (Y)
   (COND
      ((EQ Y NIL) <expression>)
       (T  <expression with recursive call to XX>)))
```

See if you can write a recursive function (called SEPARATE) which prints out each element of a list on a separate line, for example:

```
EVALUATE: (SEPARATE '(A B C (D E) F))
```

should print

```
A
B
C
(D E)
F
```

If you get stuck, try using the following guide.

```
(DEFUN SEPARATE (X)
 (COND
  ((is it a null list?) (If so do nothing))
  (otherwise (print first item of list)
  (and then SEPARATE the rest of list))))
```

Now devise a function to reverse a list, thus

```
^EVALUATE: (REVERSE '(1 2 3 4 5))

^VALUE IS: (5 4 3 2 1)
```

If you have trouble approaching the problem, imagine carrying out the reversal by hand. You might have the items written on cards and laid out on a tray. The task would be to transfer them one by one to an empty tray but reversed. The empty tray could be represented in LISP by the empty list (). Remember that CONS is the function for adding to lists. Watch out for infinite recursion! Don't forget that (CONS (CAR X) (CDR X)) is the same as X.

3.6 Property Lists

Before we present a full example LISP application, we'll take a look at a type of LISP data structure that is an extension of the basic list concept.

We have already seen that LISP's data objects are more versatile than the variables in other languages. They can take several different types of value. In addition each data object has what is called a property list, whether it has a value or is UNDEFINED. Initially this is NIL but we can extend it as much or as little as we like. It contains pairs of items: the first is always the name of the property (such as hair colour, age, weight etc.) and the second is its value (e.g. brown, 25, 11 stone etc).

Let us see how this works. Enter:

```
(PUT 'FRED 'HAIRCOLOUR 'BROWN)
```

This expression will build what is called a dotted-pair which will look like this:

```
(HAIRCOLOUR . BROWN)
```

It will also attach this pair on the property list of FRED.

Now type:

```
(PUT 'FRED 'AGE '25)
```

This produces the dotted pair

```
(AGE . 25)
```

and adds it to the property list. We can examine the property list of a data object at any time by using PLIST, for example:

```
^EVALUATE: (PLIST 'FRED)

^VALUE IS: ((HAIRCOLOUR . BROWN) (AGE . 25))
```

The real advantage of property lists, however, is that we can very easily pick out individual items, and we do this using GET, for example:

```
^EVALUATE: (GET 'FRED 'HAIRCOLOUR)

^VALUE IS: BROWN
```

Note that each property can only have one value, so if we enter:

```
^EVALUATE: (PUT 'FRED 'AGE 26)
```

the value of FRED's property AGE will now be 26.
REMPROP is used to remove a property:

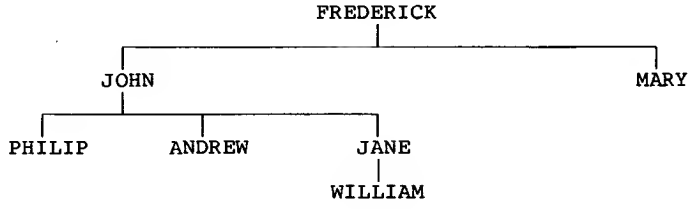
```
^EVALUATE: (REMPROP 'FRED 'AGE)

^VALUE IS: T
```

The result is T unless there is no such property, in which case it is NIL.

3.7 A Real-Life Application

Genealogy is the study of family history. We've all seen what a family tree looks like and many people find it a fascinating hobby to try and trace their own origins. LISP is ideal for helping with this sort of undertaking. Not only is it simple to build up your data structure using property lists, but in addition it takes very little effort to write programs which will interrogate the structure and give answers to questions such as 'Who was X's paternal grandfather' or 'List out all the first cousins of Y'.

We shall take a fragment of an imaginary family tree as our example.



Now let us build a segment of this family tree in LISP.

Enter the following sequence. As you go, check that what you are typing corresponds to information which you can see in the family tree. Notice also that there is nothing which represents the lines joining the names. Quite simply we give each person his or her own separate property list and note on it certain relationships to other members of the family.

Frederick's details:

```
(PUT 'FREDERICK 'SON 'JOHN)
(PUT 'FREDERICK 'DAUGHTER 'MARY)
```

John's details:

```
(PUT 'JOHN 'SON '(PHILIP ANDREW))
(PUT 'JOHN 'DAUGHTER 'JANE)
(PUT 'JOHN 'FATHER 'FREDERICK)
```

Jane's details:

```
(PUT 'JANE 'SON 'WILLIAM)
(PUT 'JANE 'FATHER 'JOHN)
```

William's details:

```
(PUT 'WILLIAM 'MOTHER 'JANE)
```

Notice that we only specify parent/child relationships. This is sufficient because we can work out the other relationships quite easily from these. For instance, two people are cousins if they have the same grandfather and grandmother.

Make sure you typed in the previous portion of the tree correctly (use PLIST to check this). Now try this:

```
^EVALUATE: (GET 'JOHN 'DAUGHTER)
```

if all is well you should get the reply:

```
^VALUE IS: JANE
```

Now let's define a function which will tell us the name of the father of an arbitrary person on the tree. We shall call the function FATHER-OF.

```
(DEFUN FATHER-OF (X) (GET X 'FATHER))
```

Type it in and then try:

a) (FATHER-OF 'JOHN)
b) (FATHER-OF 'JANE)

Check the tree to see if the answers were right. Now we can take things a stage further. We know that to say 'grandfather of Jane' means the same as 'father of father of Jane'. This can be translated directly into LISP. It becomes:

```
(FATHER-OF (FATHER-OF 'JANE))
```

So let's define a function GRANDFATHER-OF:

```
(DEFUN GRANDFATHER-OF (X) (FATHER-OF (FATHER-OF X)))
```

Type it in and then try it. You should get:

```
^EVALUATE: (GRANDFATHER-OF 'JANE)

^VALUE IS: FREDERICK
```

We have of course ignored certain difficulties. The function GRANDFATHER-OF only works out a person's grandfather on their father's side. However, we have demonstrated how simple such a function can be. You might like to work out functions for other relationships. Start with something simple like GRANDSON and then see if you can work out the functions SISTER, COUSIN etc. Where there could be more than one answer, the result should be a list.

Chapter 7 contains a much bigger example application: a program to find the shortest routes between two towns from a stored representation of the route map. Have a think about how you would approach the problem before looking at it. Also included in Chapter 7 are further instructive examples of LISP programs, many of which are functions that provide the facilities provided as standard on big LISP systems.

3.8 Conclusion

We hope that this short introduction to LISP programming has given you some idea of what the language is capable of doing. Unless you are a computer scientist, you probably still have a few qualms about being able to tackle large or complicated programs. Remember the maxim 'If in doubt, try it out!' - LISP is a language where identifying a single mistake in your logic can result in a huge leap in understanding which carries over to many other problems. Things really do get easier with practice and the results are often very worthwhile because LISP encourages you to think of applications not usually connected with computing. For example, one medical student used a data-structure representing the way that human bones are connected together as a revision aid for his examinations, and a puzzle fanatic solved newspaper problems with the aid of LISP programs. We hope you find some equally novel uses for the language!

4 Implementation

4.1 Atoms

ATOM LISP has four types of atom:

1	Number atoms, e.g.	6 -124 31695
2	Character atoms, e.g.	CAMBRIDGE FIELD26 !"#$%&
3	Subr atoms, e.g.	PLUS CAR PRINT
4	Fsubr atoms, e.g.	LOOP WHILE COND

4.1.1 Number Atoms

Numbers can have values from -32768 to 32767. Only integers are available. The evaluation rule for numbers is simple. These atoms always evaluate to themselves. For example, the value of the atom -21 is -21.

Numbers between 32767 and -32768 are converted to number atoms. Numbers outside this range will be returned as unconverted character atoms.

4.1.2 Character Atoms

Character atoms can have names containing from 0 to 249 characters. There is no restriction on the characters (within the ASCII character set). They are used as identifiers ('printnames') of data objects. Certain character atoms cannot be typed directly because they contain blanks, full stops, dollar signs or parentheses. A special escape combination is provided for these non-standard atoms. The construction for the character string XYZ is $$%XYZ%. The two dollar signs are compulsory. The % characters at each end of the string could have been any typable character except RETURN. There is no way to type a character atom which contains RETURN.

4.1.3 Special Character Atoms

There are four special character atoms: NIL, T, LAMBDA and UNDEFINED. If the values of any of these are altered then the LISP system could fail. They should only be used for the special purposes given here.

NIL has two uses. It is used to terminate lists and as the value 'FALSE' in logical expressions. The value of NIL is always NIL and it cannot have any other properties. F is a synonym for NIL intended for use in logical expressions.

T represents 'TRUE' in logical expressions. Its value is T. Beware

- it is easy to forget that T is special and use it as a variable name, with frequently disastrous results.

UNDEFINED has the value UNDEFINED. It has a non-standard property list to save it from removal by the Garbage Collector. It can not have any normal properties.

LAMBDA is the flag word used to introduce function definitions. Its value is LAMBDA, but this is not usually very important. It can have properties.

Data objects may also have properties other than the value. New properties may be added using PUT and referenced by GET. Undefined data objects have the value UNDEFINED. This is a rather special state which means that the object does not appear on the object list. It will also be liable to be removed from the LISP system by the Garbage Collector unless it is referenced by some data structure.

4.2 Lists

Standard lists are formed from a left bracket '(' followed by the members of the list followed by a right bracket ')'. The members can be atoms or can be lists themselves. There is no limit to the number of members a list can have, other than the amount of computer memory available.

Before we go on to non-standard lists it is necessary to look at how LISP stores lists. LISP manipulates data using pointers to the item rather than the data itself. The pointer is just the address of the item in memory. Lists are built up from pointer pairs. At a certain point in a list the first pointer of the pair (CAR) points to the current member of the list and the second pointer (CDR) to the next pointer pair in the list. The CDR of the last pointer pair in a list is specially marked - it is NIL.

If the CDR pointer points to an atom then we have a 'dotted pair'. This is printed out by LISP as, for example:

```
(A . B)
```

Dotted pairs can appear at the end of a list, for example:

```
(A (LIST) ENDING IN A . PAIR)
```

4.3 Functions

When LISP is evaluating a list, it assumes that the first atom it meets after the opening bracket will be a function definition. If not, it will evaluate anything it finds there up to twice in an attempt to reach a valid function. For example in

```
(PLUS X 3)
```

The character atom PLUS is not a function definition. However, its value is a Subr atom which is a function definition. The four types of function definition are:

1 Subr atom
2 Fsubr atom
3 Expr LAMBDA expression
4 Fexpr LAMBDA expression

4.3.1 Subrs and Fsubrs

Subrs and Fsubrs are LISP system functions written in machine code. The LISP interpreter always evaluates all the arguments of a Subr. It then applies the Subr code to these values.

Some special functions have arguments which cannot be evaluated (e.g. COND) or must have their arguments unevaluated (e.g. QUOTE). These are coded as Fsubrs. The interpreter simply hands them the list of unevaluated arguments.

It is not possible to create or reference a Subr or Fsubr atom using a READ. Typing FSUBR#4593 for example, would simply create a character atom.

4.3.2 Exprs and Fexprs

Expr's and Fexpr's are functions written in LISP. Programming in LISP primarily consists of defining new LISP functions. The way to do this is to make a LAMBDA expression the value of the name of the function. A LAMBDA expression has the following syntax:

```
(LAMBDA parameterdefinition action 1 action 2 ... )
```

The action parts of the LAMBDA expression are evaluated one by one and the last one is returned as the value of the function.

Normally we want to define functions with arguments. In the case of an Expr the parameter definition is a list of variable names. Here is a function to give an atom a colour property:

```
(SETQ PUTCOLOUR
        '(LAMBDA (ATM HUE) (PUT ATM 'COLOUR HUE)))
```

When a PUTCOLOUR expression is evaluated, the arguments will first be evaluated one by one. These values are assigned to the corresponding variable names in the LAMBDA expression. For example, if we evaluate

```
(PUTCOLOUR 'BANANA 'YELLOW)
```

then during the evaluation of PUTCOLOUR, ATM has the value BANANA and HUE has the value YELLOW.

Normally there must be at least as many arguments in the function call as there are variable names in the LAMBDA expression. However, an Expr parameter list can also contain optional parameters. These are indicated by placing the variable names in brackets. Optional parameters must come after all the simple parameters in the parameter list. If no argument is provided to match the optional parameter no error will be signalled; the variable will just take the value NIL. This default can be changed by turning the optional parameter in parentheses into a dotted pair. The default value now becomes the CDR of the dotted pair. Consider this function called PRINT2TO5 :

```
(LAMBDA (A B (C) (D . 0) (E IS A LIST)) (PRINT A B C D E))
```

In this example:

A and B are simple parameters.
C is an optional parameter with default value NIL.
D is an optional parameter with default value 0.
E is an optional parameter with default value (IS A LIST).

Optional parameter variables can be used when your program requires

variables which are local to a function.

Finally, Fexprs are used where the arguments of the function must not be evaluated. The parameter description consists of a single variable name. The unevaluated argument list of the function is assigned to this variable when the function is called. For example a function ASSIGN having the same effect as SETQ could be defined in this way:

```
(SETQ ASSIGN
'(LAMBDA Q (SET (CAR Q) (EVAL (CADR Q)))))
```

The Exprs and Fexprs provided by ATOM LISP provide a useful range of examples of LISP functions. To obtain a listing, type the name of the Expr or Fexpr in response to the '^EVALUATE:' prompt.

4.4 Garbage collection

During the course of a normal LISP program, data structures are constantly being added to and trimmed. The computer has to be able to maintain a pool of data storage locations (called free storage) from which it can draw. Every so often it becomes necessary to tidy up the discarded locations and return them in an orderly fashion to free storage.

The LISP system automatically brings in a program called the Garbage Collector whenever it detects a lack of working space. The user can cause a garbage collection at any time by calling on the function RECLAIM. For example:

```
^EVALUATE: (RECLAIM)
```

Programs which are near to the limit of available memory are likely to be slowed down to a small degree by repeated garbage collections. With (MESSON 3) the number of reclaimed bytes is shown and so it is also a good way of finding how much memory is still available.

4.5 Printing

PRINT and PRIN0 print expressions in a standard format:

Character atoms	- printed character by character
Number atoms	- leading zeros suppressed, no leading blanks printed
Subr/Fsubr atoms	- SUBR# or FSUBR# followed by the decimal entry address of the machine code
Lists	- elements surrounded by a pair of brackets and separated by spaces

SPRINT formats lists to make them more readable. The variable LINEWIDTH contains the number of characters per line assumed by PRINT. This can be set to suit the output device being used. Its default value is 31 to suit the ATOM screen.

4.6 System Routines

For completeness, we give a list of all those functions and variables

which have an effect on the system in addition to their role as ordinary LISP entities. Details of any not described in this chapter will be found in the Glossary.

CALL, ERROR, ERRORSET, LINEWIDTH, LOAD, MESSOFF, MESSON, OBLIST, PEEK, POKE, READ, RECLAIM, SAVE

4.7 Differences from Other LISPs

Atom LISP has no PROG and GO functions. In many other LISPs, PROG has two logically separate functions:

1 Defining local variables
Atom LISP does this in the LAMBDA parameter list. Local variables are placed in parentheses after the normal function parameters. See Section 4.3.2.

2 Allowing iterative programs
Atom LISP does this through its LOOP, WHILE, UNTIL construction.

4.8 Using Extra Memory

Extra RAM can be added to the ATOM, contiguous with the lower text space, giving more space for the LISP image and stack. On start-up, the system automatically initialises the stack at the limit of contiguous RAM at or above #3C00 (see the Memory Map in Appendix A). Thus when you upgrade your ATOM hardware, LISP will automatically make full use of the extra memory.

5 Disc Input/Output

In addition to the LOAD and SAVE commands described in Chapter 2, ATOM LISP has a set of functions for use with the DOS only.

5.1 OPEN

The OPEN function opens the named file for input or output. Its form is:

(OPEN <filename> <mode>)

If mode is T this indicates that the file already exists. If mode is NIL then a new file will be created. OPEN returns a value as do all LISP functions. The value in this case is the 'handle' of the file. This is a code number which the system allocates temporarily as an identifier to the file. It is required by all the functions set out below and so, to avoid having to remember it, you should use OPEN in conjunction with SETQ. For example,

(SETQ H (OPEN 'DATA NIL))

This opens a new file called DATA and assigns the value of the handle to H. We can now use H in any function where DATA's handle is required.

A maximum of five files may be open at a given time although more can be accessed if necessary by using the CLOSE command for those temporarily not needed.

5.2 CLOSE

The CLOSE function closes a current file in a tidy way and if necessary writes an EOF (end-of-file) marker. Its form is:

(CLOSE <handle>)

For example:

(CLOSE H)

closes the file whose handle is H. If the handle is 0, all current files are closed whatever their handles. If a file is closed and then re-opened, it will be given a new handle. This should be reassigned using SETQ.

5.3 Input Functions

There are three input functions for use with DOS files:

GETCHAR	returns the next character as a character atom
READ	returns an entire LISP expression (atom or list)
READLINE	returns all the characters upto the next carriage return as a single character atom

They are all used in the form:

(<function> <handle>)

The handle specifies the file, which must be open when the input function is used. For example,

```
^EVALUATE: (SETQ H (OPEN 'TEXT T))

^VALUE IS: 32

^EVALUATE: (SETQ CHAR (GETCHAR H))

^VALUE IS: A            (if A is the first character of TEXT)
```

Any of these functions can be used without a handle, or with a handle of 0, to input from the keyboard, for example:

```
(SETQ EXP (READ 0))
```

The EOF function detects whether an end-of-file marker has been reached when reading. Its form is:

(EOF <handle>)

It returns T if end-of-file has been reached and NIL if it has not.

5.4 Output Functions

In addition to PRINT, there are two output functions for use with the DOS. They are:

WRITE0	writes the given items to the specified file
WRITE	writes the given items to the specified file, preceeded by a carriage return

Their form is:

(<function> <handle> <items to be printed>)

For example:

```
(WRITE H 'TEXT 'AREA 'FULL)
```

A file handle of 0 causes the output to be sent to the screen, so

```
(WRITE 0 X)     is equivalent to     (PRINT X)
```

and

```
(WRITE0 0 X)    is equivalent to     (PRINT0 X)
```

The READ command requires each record it is reading to be terminated with a carriage-return character. Thus when using WRITE or WRITE0 you should finish by using (WRITE <handle>) or (WRITE0 <handle> CR) either of which will send a single carriage-return to the screen.

Finally, a suitable sequence for printing the contents of file IMAGE1 would be:

```
(SETQ H (OPEN 'IMAGE1 T))
```

followed by:

```
(LOOP
   (UNTIL (EOF)
   (PRIN0 (GETCHAR H)))

(CLOSE H)
```

6 The Editor

The LISP editor is itself written as a LISP function. A listing is given in the Glossary. Below are summarised the editing commands followed by an example of their use. The Editor can be used on any LISP expression but its main purpose is for making changes whilst developing function definitions.

6.1 Editor Commands

The full set of editor commands is as follows:

Command	Function
A	Finds CAR of current expression.
B	Restores previous current expression (but retaining any change made).
C	CONSes the item which is next typed onto the current expression.
D	Finds CDR of current expression.
R	Replaces current expression with the next expression typed.
X	Deletes first item from current expression.
RETURN	Prints out current expression (provided another command is not in effect at the time).

Where a command takes an expression, a RETURN will be printed and the expression is then entered, terminated by RETURN. The effect of the operation is not automatically displayed - the RETURN command can be used to show this.

An asterisk (*) is printed when a NULL item is reached. It often indicates that A or D has been used inappropriately. The editor responds to unrecognised commands with a query (?).

6.2 An Editing example

Let us imagine that we wish to use the editor to change the expression:

```
(ONE (TWO 3) 4 XY (5 SIX))
```

into

```
(1 (2 3) 4 (5 6) 7)
```

We shall first of all work through the edit step by step explaining as we go along. Finally we give an actual listing showing how the edit would be done in reality.

First set X to the list that requires editing.

```
^EVALUATE: (SETQ X '(ONE (TWO 3) 4 XY (5 SIX)))

^VALUE IS: (ONE (TWO  3) 4 XY (5 SIX))
```

In the present rather artificial example this step would not strictly be necessary but in real life, expressions to be edited will almost invariably have been assigned to some data object - in this case we use X.

Now enter:

```
^EVALUATE: (SETQ TEMP (EDIT X))
```

This sequence states that we should enter the editor and that it is to use the value of the atom X as its argument. The expression which results from the completed editing sequence will be assigned to the data object TEMP. This is a safety measure; we could have assigned the result directly back to X but this loses the original list. In the event of making a mistake during editing we would not be able to restore things to their original state.

The Editor will reply to the above like this:

```
(ONE (TWO 3) 4 XY (5 SIX))
```

Let us now commence the actual editing process. Enter:

```
A
```

This directs the editor to find the CAR of the current expression, i.e. ONE. Now enter R <RETURN> followed by 1 <RETURN>. This replaces the current expression, 'ONE', by '1'. In order to check that this has been done correctly, we can use B to move back to the full list:

```
B <RETURN>
```

This turns the entire result of the edit so far into the current expression. You should get the reply:

```
(1 (TWO 3) 4 XY (5 SIX))
```

So far, so good. Now let's change 'TWO' into '2'. We can see that TWO is obtained by taking the CDR, followed by the CAR, followed by the CAR of the current expression, so enter:

```
DAA
```

We typed D before AA because it was necessary to find the CDR before we could find its CAR.

Enter R <RETURN> followed by 2 <RETURN>, followed by BB. R 2 replaces 'TWO' by '2'. We typed B twice so as to work our way back through the nested levels of the argument X.

Now move down to XY by typing DD:

```
(XY (5 SIX))
```

Then enter X to delete the first expression XY to give:

```
((5 SIX))
```

We now need to change 'SIX' into '6'. Keep entering D until you get

```
((5 SIX))
```

Now we wish to look at the first (and only) item of this list, so enter:

A

to get (5 SIX). SIX is obtained by taking the CDR followed by the CAR of this expression so enter:

D followed by A

then enter:

R followed by 6 <RETURN>

All that remains now is to insert the value 7 at the end of the list. Use B three times to do this. Then enter:

D

to get NIL. Remember that in LISP, NIL is equivalent to the empty list (). We now need to add our 7 to this and then work back through the structure. Enter:

C followed by 7

Now use B repeatedly until you get back to the EVALUATE prompt. TEMP will now hold the edited expression.

Here is the uncommented version of the editing session we have just worked through. User input is underlined.

```
^EVALUATE: (SETQ X '(ONE (TWO 3) 4 XY (5 SIX)))

^VALUE IS: (ONE (TWO 3) 4 XY (5 SIX))

^EVALUATE: (SETQ X2 (EDIT X))

(ONE (TWO 3) 4 XY (5 SIX))
A
ONE
R
1
B
(1 (TWO 3) 4 XY (5 SIX))
DAA
TWO
R
2
BB
((2 3) 4 XY (5 SIX))
DD
(XY (5 SIX))
X
((5 SIX))
ADA
SIX
R
6
BBB
((5 6))
D
```

```
NIL
C
7
BB
(4 (5 6) 7)
BBB
^VALUE IS: (1 (2 3) 4 (5 6) 7)
```

Note that since the Editor is written in LISP you can use it to edit itself, putting in extra commands or changing the names of existing ones to suit yourself. As medium-sized LISP exercises you may like to try to extend the Editor to provide a command Q (for example) that is equivalent to B's in sufficient quantity to get you to the top of the list being edited, and some form of F command that allows you to find sublists by matching simple patterns.

7 Further Examples

7.1 Numeric Functions

ABS

Returns the absolute value of a number

```
(DEFUN ABS (X) (COND ((MINUSP X) (DIFFERENCE 0 X)) (TT X)))
```

MAX

Returns the largest number in a list

```
(DEFUN MAX
   (X (N . -32767))
   (LOOP
      (WHILE X N)
      (COND ((GREATERP (CAR X) N) (SETQ N (CAR X))))
      (SETQ X (CDR X))))
```

MIN

Returns the smallest number in a list

```
(DEFUN MIN
   (X (N . 32767))
   (LOOP
      (WHILE X N)
      (COND ((LESSP (CAR X) N) (SETQ N (CAR X))))
      (SETQ X (CDR X))))
```

NCR

Returns the number of combinations ${}^{n}Cr$.

```
(DEFUN NCR
   (N K)
   (COND
      ((EQ ZEROP K) 1)
      ((EQ K N) 1)
      (T (PLUS (NCR (DIFFERENCE N 1) (DIFFERENCE K 1))
               (NCR (DIFFERENCE N 1) K)))))
```

NCR takes two numeric arguments, N and R, and returns the number of combinations of N objects taken R at a time. R must be less than or equal to N. The recursive definition is as follows:

$$
\begin{aligned}
{}^{n}Cr\ [n,r] &= 1 && \text{if } r=0 \\
&= 1 && \text{if } r=n \\
&= {}^{n}Cr\ [n-1,r] + {}^{n}Cr\ [n-1,r-1] && \text{otherwise}
\end{aligned}
$$

This is based on Pascal's triangle.

PFR

Discovers the prime factors of a number. PFR uses the auxilliary function PF1.

```
(DEFUN PFR
   (N)
   (COND
      ((ZEROP (REMAINDER N 2))
         (CONS 2 (PFR (QUOTIENT N 2))))
      (T (PF1 N 3))))

(DEFUN PF1
   (N M)
   (COND
      ((EQ N 1) NIL)
      ((LESSP N (TIMES M M)) (LIST N))
      ((ZEROP (REMAINDER N M)) (CONS M (PF1 (DIVIDE N M) M)))
      (T (PF1 N (PLUS M 2)))))
```

7.2 List Processing Functions

APPEND

Concatenates two lists.

```
(DEFUN APPEND
   (X Y)
   (COND ((NULL X) Y) (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

Note the difference between APPEND and CONS:

```
^EVALUATE:  (APPEND '(A B C) '(D E F))

^VALUE IS: (A B C D E F)

^EVALUATE: (CONS '(A B C) '(D E F))

^VALUE IS: ((A B C) D E F)
```

NCONC

Concatenates two lists like APPEND, but does not copy the first list.

```
(DEFUN NCONC
   (X Y (Z))
   (SETQ Z X)
   (COND
      ((NULL X) Y)
      (T (LOOP (WHILE (CDR X) (RPLACD X Y) Z) (SETQ X (CDR X))))))
```

DELETE

Deletes all occurences of X from the list Y.

```
(DEFUN DELETE
   (X Y)
   (COND
      ((NULL Y) NIL)
      ((EQ X (CAR Y)) (DELETE X (CDR Y)))
      (T (CONS (CAR Y) (DELETE X (CDR Y))))))
```

EQUAL

Tests two lists for equality. NIL is returned if they are equal, and T otherwise.

```
(DEFUN EQUAL
   (X Y)
   (COND
      ((EQ X Y))
      ((OR (ATOM X) (ATOM Y)) NIL)
      ((AND (EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y))))))
```

Note that EQUAL tests two lists for equality right down to the constituent atoms, whereas EQ just tests whether they are in fact the same list. For example:

```
^EVALUATE: (SETQ X '(A B C))

^VALUE IS: (A B C)

^EVALUATE: (SETQ Y '(A B C))

^VALUE IS: (A B C)

^EVALUATE: (EQ X Y)

^VALUE IS: NIL

^EVALUATE: (EQUAL X Y)

^VALUE IS: T
```

MEMBER

Test whether atom X is a member of list Y, returning T if so and NIL otherwise.

```
(DEFUN MEMBER
   (X Y)
   (LOOP (WHILE Y) (UNTIL (EQ X (CAR Y)) T) (SETQ Y (CDR Y))))
```

For example:

```
^EVALUATE: (MEMBER 'A '(C D A M))

^VALUE IS: T
```

REMOB

Deletes a data object from the object list by removing all its properties and setting its value to UNDEFINED.

```
(DEFUN REMOB
   (X)
   (SETQ X (QUOTE UNDEFINED))
   (MAP (QUOTE (LAMBDA (Y) (REMPROP X (CAR Y)))) (PLIST X)))
```

REVERSE

Reverses a list.

```
(DEFUN REVERSE
   (X (Z))
   (LOOP (WHILE X Z) (SETQ Z (CONS (CAR X) Z)) (SETQ X (CDR X))))
```

SUBST

Replaces all occurences of Y in list Z to X

```
(DEFUN SUBST
   (X Y Z)
   (COND
      ((EQ Y Z) X)
      ((ATOM Z) Z)
      (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))))
```

MAP

Applies the given function to each element of the list and returns the result of the last application.

```
(DEFUN MAP
  (FUN LIS)
  (COND
  ((NULL LIS) NIL)
      (T (FUN (CAR LIS)) (MAP FUN (CDR LIS)))))
```

MAPC

Applies a function to the elements of a list like MAP, but returns a list of the results.

```
(DEFUN MAPC
   (FUN LIS)
   (COND
      ((NULL LIS) NIL)
      (T (CONS (FUN (CAR LIS)) (MAPC FUN (CDR LIS))))))
```

GENSYM

Generates a unique single-character atom for use where new data objects are required but their names are unimportant.

```
(DEFUN GENSYM
   NIL
   (CHARACTER
      (SETQ
         QGEN
         (COND ((NOT (NUMBERP QGEN)) 128) (T (PLUS 1 QGEN))))))
```

TRACE

Modifies the definition of a fuction so that a message is printed whenever the function is used, as an aid to debugging. For example, after

```
 ^EVALUATE: (TRACE PLUS)
```

every use of PLUS will print

```
#(PLUS <args>)
```

where <args> is the list of arguments that were supplied to PLUS.
To turn the TRACE facility off, enter

```
(UNTRACE PLUS)
```

Note that TRACE should not be used on any of the functions used by TRACE (for obvious reasons!), namely:

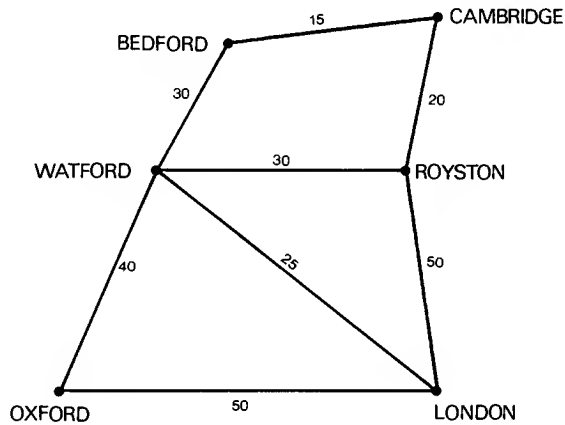
CAR, CDR, EVAL, PRINT, QUOTE, COND, SET, PUT, GET.

TRACE requires SUBST as defined in this section. The definitions of TRACE and UNTRACE are as follows:

```
(DEFUN TRACE
   X
   (PUT (CAR X) (QUOTE OLD) (EVAL (CAR X)))
   (SET
      (CAR X)
      (SUBST
         (CAR X)
         (QUOTE ?)
         (QUOTE
            (LAMBDA
               Y
               (PRINT (QUOTE #) (CONS (QUOTE ?) Y))
               (EVAL (CONS (GET (QUOTE ?) (QUOTE OLD)) Y)))))))

(DEFUN UNTRACE
   X
   (SET (CAR X) (GET (CAR X) (QUOTE OLD)))
   (REMPROP (CAR X) (QUOTE OLD)))
^VALUE IS: )
```

7.3 Route Map Example

This example program uses a stored representation of the following map to determine the shortest route between any two of the towns:



For each town on the map there is an association list of distances to the other town that can be reached directly. Thus for WATFORD the association list is:

```
((BEDFORD . 30) (ROYSTON . 30) (LONDON . 25) (OXFORD . 40))
```

This exists as the value of WATFORD's property called NEIGHBOURS. It is returned by:

```
^EVALUATE: (GET 'WATFORD 'NEIGHBOURS)
```

The program and data are supplied on the LISP cassette as the file IMAGE. This should be loaded as described in Section 2.2. To find the length of the shortest route between any two of the towns on the map, use the function DIST with the qouted names of the towns as the arguments, for example:

```
^EVALUATE: (DIST 'CAMBRIDGE 'OXFORD)

DISTANCE 85 MILES VIA:
CAMBRIDGE
BEDFORD
WATFRD
OXFORD
^VALUE IS: NIL
```

The function definitions are as follows:

```
NPUT
(LAMBDA (A B) (PUT A (QUOTE NEIGHBOURS) B) A)

MEMBER
(LAMBDA
   (A B)
   (LOOP (WHILE B) (UNTIL (EQ A (CAR B))) (SETQ B (CDR B))))

TREEADD
(LAMBDA
   (X TREE)
   (COND
      ((NULL TREE) (CONS X (CONS NIL NIL)))
      ((LESSP (CAR X) (CAAR TREE))
         (RPLACA (CDR TREE) (TREEADD X (CADR TREE)))
         TREE)
      (T (RPLACD (CDR TREE) (TREEADD X (CDDR TREE))) TREE)))

FLATTREE
(LAMBDA
   (TREE L)
   (COND
      ((NULL TREE) L)
      (T
         (FLATTREE
            (CADR TREE)
            (CONS (CAR TREE) (FLATTREE (CDDR TREE) L))))))
```

```
NEXT-CITY
(LAMBDA
   (FINISH ROUTE-SO-FAR END-ROUTE (NL) (OTHER-ROUTES) (NEW-ROUTE))
   (SETQ NL (GET (CADR ROUTE-SO-FAR) (QUOTE NEIGHBOURS)))
   (LOOP
      (WHILE NL (CONS END-ROUTE OTHER-ROUTES))
      (SETQ
         NEW-ROUTE
         (CONS
            (PLUS (CDAR NL) (CAR ROUTE-SO-FAR))
            (CONS (CAAR NL) (CDR ROUTE-SO-FAR))))
      (COND
         ((AND
            (LESSP (CAR NEW-ROUTE) (CAR END-ROUTE))
            (NOT (MEMBER (CAAR NL) (CDR ROUTE-SO-FAR))))
            (COND
               ((EQ (CAAR NL) FINISH) (SETQ END-ROUTE NEW-ROUTE))
                     (T (SETQ OTHER-ROUTES (CONS NEW-ROUTE
OTHER-ROUTES))))))
      (SETQ NL (CDR NL))))

SORT
(LAMBDA
   (L LIMIT-DISTANCE (TREE))
   (LOOP
      (WHILE L (FLATTREE TREE NIL))
      (COND
         ((LESSP (CAAR L) LIMIT-DISTANCE)
            (SETQ TREE (TREEADD (CAR L) TREE))))
      (SETQ L (CDR L))))

NCONC
(LAMBDA
   (X Y (Z))
   (SETQ Z X)
   (COND
      ((NULL X) Y)
      (T (LOOP (WHILE (CDR X) (RPLACD X Y) Z) (SETQ X (CDR X))))))

ADD-TO-PLIST
(LAMBDA
   (DIST TOWN1 TOWN2)
   (PUT
      TOWN1
      (QUOTE NEIGHBOURS)
      (CONS (CONS TOWN2 DIST) (GET TOWN1 (QUOTE NEIGHBOURS))))
   (PUT
      TOWN2
      (QUOTE NEIGHBOURS)
      (CONS (CONS TOWN1 DIST) (GET TOWN2 (QUOTE NEIGHBOURS)))))

ROUTE
(LAMBDA
   (START FINISH (RSFL) (SHORTEST 32767) (X))
   (COND
      ((EQ START FINISH) (LIST 0 FINISH START))
      (T
         (SETQ RSFL (LIST (LIST 0 START)))
         (LOOP
            (SETQ RSFL (SORT RSFL (CAR SHORTEST)))
```

```
            (WHILE RSFL (COND ((CDR SHORTEST) SHORTEST)))
            (SETQ X (NEXT-CITY FINISH (CAR RSFL) SHORTEST))
            (SETQ RSFL (NCONC (CDR RSFL) (CDR X)))
            (SETQ SHORTEST (CAR X))))))

DIST
(LAMBDA
   (T1 T2 (RT))
   (SETQ RT (ROUTE T2 T1))
   (PRINT (QUOTE DISTANCE) (CAR RT) (QUOTE MILES VIA:))
   (LOOP (SETQ RT (CDR RT)) (WHILE RT) (PRINT (CAR RT))))
```

8 Error Handling

8.1 Traceback

There are certain things that LISP finds it impossible to do, such as generating a number larger than 32767 or finding the CAR of an atom. When one of these is attempted, LISP normally halts and prints some information about what went wrong. For example, if we try to use XTAB with a non-numeric argument, we would get the following:

```
^EVALUATE : (XTAB 'NOTNUMBER)
^ERROR NUMBER 21
^ARG: NOTNUMBER
^ARG: (DIFFERENCE S 1)
^ARG: (SETQ S (DIFFERENCE S 1))
^ARG: (MINUSP (SETQ S (DIFFERENCE S 1)))
^ARG: (UNTIL (MINUSP (SETQ S (DIFFERENCE S 1))))
^ARG: (LOOP (UNTIL (MINUSP (SETQ S (DIFFERENCE S 1)))))
(PRIN0 BLANK))
^S = NOTNUMBER
^ARG: (XTAB 'NOTNUMBER)

~EVALUATE :
```

The first line of the error message tells us that this error is error type 21. The various error numbers are described at the end of this chapter. Error 21 means that the arguments are not all numeric where this was expected. The remainder of the printout tells us exactly what was going on when the error occurred. Each line labelled 'ARG:' shows an expression which LISP is evaluating. These appear in reverse chronological order: the last ARG is the one that LISP was originally trying to evaluate and the others are ones that LISP had to evaluate to do this. In the example we see in the third line

```
(DIFFERENCE S 1)
```

It is the expression that actually caused the error as LISP could not subtract one from the character atom NOTNUMBER. The fourth line shows that LISP was evaluating

```
(DIFFERENCE S 1)
```

as a sub-expression of

```
(SETQ S (DIFFERENCE S 1))
```

The following ARGs give progressively higher level expressions.

The top level expression (just before the S= line) is part of the definition of the XTAB function. When we reach this point at which the XTAB function was called we have:

```
^S=NOTNUMBER
```

This tells us that when the error ocurred the variable S had the value NOTNUMBER. In general all the local variables of a function would be printed here. The final ARG shows the cell to XTAB. If XTAB has been called by another function the printout would have gone on to deal

with that.

This summary of events at the time of the error is called the traceback. You can control whether various parts of it are printed or not by using the MESSON and MESSOFF functions which are described in the Glossary. Page mode is automatically set during an error traceback, so the output will stop and wait for the space bar to be pressed when it has filled the screen.

8.2 Error-Handling Functions

There are two built-in LISP functions to help in handling errors: ERROR and ERRORSET. They are described separately in the Glossary. ERROR prints a message and then triggers the traceback mechanism.

ERRORSET is the more interesting of the two. Normally when an error occurs, the traceback works through all the currently active functions and gives a new EVALUATE prompt. ERRORSET 'catches' the traceback on its way past and allows a program to decide for itself what to do with an error.

8.3 Error numbers

Number	Description
0	**No space for stack**
1	**No space for variable binding**
2	**No space for new cell**

These three error messages mean that LISP has filled the available workspace and has carried out a garbage collection but could not free any space. This situation can arise for a number of reasons:

a) The program needs more memory than is available.
b) The program is in a loop, repeatedly allocating more space.
c) The workspace is filled with a large data structure or program from earlier work .

No traceback is printed because LISP would not be able to find the workspace needed to print it. Some suggested solutions are:

a) Remove all unnecessary programs and data structures by setting the relevant variables to UNDEFINED - in particular remove OLDDEF properties from edited functions.
b) Look through the program for recursive calls which could be replaced by LOOP - this uses less memory.
c) Reduce the size of the data structures in the program.
d) Check that the logic of the program is correct.

3 Not enough arguments for an Fsubr

This has to be caused by an incorrectly written program. In the traceback the second ARG will show the offending expression. Check in the Glossary how many arguments the function should have and edit the program.

4 Interrupt during evaluation

You pressed the ESC key on the keyboard. The traceback will show what was happening at the time.

5 Expression in function position not a function

The first member of a list to be evaluated should be a function definition. The most common way of obtaining this error is to forget to define the function. LISP will evaluate a function up to twice to try to come up with a Subr atom, Fsubr atom or list beginning with LAMBDA.

The first ARG in the traceback shows what LISP had reached when it gave up trying to produce a function. The second ARG shows the list being evaluated. The first member of this is the invalid function.

6 Wrong number of arguments for Expr or Subr

There are a number of ways this error can arise:

a) The program gives more than 28 arguments for the function.
b) There are less than the minimum number of arguments required for the Subr.
c) There are not enough arguments to satisfy all the compulsory parameters of the Expr.

The second ARG of the traceback shows the offending expression. This error can easily arise for reason (a) in an APPLY expression.

7 Syntax error in LAMBDA expression

This error can occur either while binding the arguments of an Expr or Fexpr or while executing it. In either case, check the definition of the function against the rules in Section 4.3.2. The second ARG of the traceback will show the incorrectly formed function as the first member of the list.

8 Syntax error in READ
9 Dotted pair syntax error in READ

The s-expression input to a READ function was badly formed. The things which can go wrong here are:

a) There was an initial full stop or right parentheses
b) A $ in an escape sequence was not immediately followed by another $.
c) $$ was immediately followed by RETURN.
d) The full stop of a dotted pair was not followed by a single expression and close bracket.

If you were typing on the keyboard the evidence will still be on the screen. If the input was from the disc file then the badly formed expression may take more tracking down. Beware of printing non-standard atoms onto a disc file. If it was necessary to use the $$ escape sequence to enter the atom it may not read correctly from a file if it had been printed there in the normal way.

10 String too long

This is a somewhat unlikely error which could happen during a READ. LISP strings can contain up to 249 characters. If an atom longer than this is typed this error will arise.

11 Interrupt during PRINT

You have pressed the ESC key while LISP was printing an expression. If LISP starts to print a circular list it is useful to be able to stop the printout. There is no traceback - it would only have started to print the same list again!

12 Attempt to print unknown atom type

This error message really means that LISP has crashed. It will only occur if:

a) The LISP system or a LISP program has loaded badly
b) You have altered the value of a special atom like T

13 Syntax error in COND expression

This is a programming error - a COND expression has been written incorrectly. This error arises when there is an atom where a condition/action list was expected. The first ARG will show the atom and the second ARG the incorrect COND expression. Check the expression against the syntax given in the Glossary.

14 Attempt to take CAR or CDR of an atom

CAR and CDR and their compounds can only be applied to lists or dotted pairs. The first ARG of the traceback will show the offending atom. In a compound function like CADR this may be an intermediate result rather than the original argument. The solution is to review the logic of the program to ensure that CAR and CDR are never applied to an atom.

15 LISP ERROR function called

The ERROR function generates this error number and corresponding traceback. The message printed by ERROR will presumably give some clue as to what went wrong.

16 Attempt to assign a value to other than a data object

Only data objects can be assigned values. If a SET or SETQ function has any other type of object as its first argument, this error will arise. The second ARG in the traceback shows the expression in which this occurred.

If this happens in a SETQ expression it is a programming error: make sure the first argument is the character atom name of a data object. If it occurs in a SET expression the logic of the program may be at fault.

17 Not character atom where expected

This error can have a variety of causes. One of the arguments (usually the first) of an expression should have evaluated to a character atom, but did not.

The second ARG of the traceback gives the expression in which the error arose.

18 . Arithmetic overflow

An arithmetic expression has produced a result greater than 32767 or less than -32768, or there was an attempt to divide by zero. The first ARG of the traceback normally gives the value of the last argument of the offending expression. The expression is shown in the second ARG.

19 First argument of RPLACA or RPLACD not a list

The first argument of RPLACA and RPLACD must be a list or dotted pair. The first ARG of the traceback gives the value of the second argument of the function. The second ARG gives the RPLACA or RPLACD expression.

20 Property list syntax error

This error occurs when a property list does not have the following form:

((propname . property) (propname . property) ...)

The error usually means that the non-standard property list of UNDEFINED has been accessed or the property list has been corrupted by the use of RPLACA or RPLACD at some point.

21 Arguments not all numeric

Certain functions expect all their arguments to be number atoms. If any argument is not, this error occurs. The first ARG shows the value of the last argument of the incorrect expression. The expression itself is shown in the second ARG.

22 Function in APPLY not Expr or Subr

The first argument of APPLY must evaluate to a Subr atom or an Expr list. This condition is stricter than that for an expression in the function position of a list and it is easy to make an error.

23 Bad argument list for APPLY

The second argument of APPLY must be a list. This error occurs when it is not. The first ARG gives the incorrect second argument. The second ARG shows the expression in which it occurred.

24 For future expansion

25 For future expansion

Glossary

This reference section lists all the atoms available initially on the LISP object (OBLIST). More can be added by using DEFUN or SETQ to define new variables. For each atom an explanation of its value and use is given. In the case of functions, the explanation includes whether it is a Subr, Fsubr, Expr or Fexpr and how the function works. All Subrs and Exprs evaluate all their arguments before applying the function. This is not repeated in each explanation. Fsubrs and Fexprs are always fully explained.

Character **atoms**

carriage return
space
$
(
)
.

These are single character atoms which are difficult to type into LISP expressions. Each one has itself as its value. They can be accessed from the following atoms:

Atom	Value
CR	return
BLANK	space
DOLLAR	$
LPAR	(
RPAR	)
PERIOD	.

For example:

```
(PRINT DOLLAR 35)
```

will print

```
$35
```

AND **Fsubr**

(AND predicate predicate ...)

AND can have any number of arguments. It returns T if and only if the values of all its arguments are non NIL. Otherwise it returns NIL. AND does not necessarily evaluate all its arguments. It goes through the list evaluating them one by one until:

a) the value of an argument is NIL - the value returned is then NIL.
b) the end of the list is reached - the value returned is T.

For example:

```
(AND 'TEA 'CAKES)
```

has the value T

```
(AND (LISTP X) (CDR X))
```

is T if X is a list of at least one member. In this example (CDR X) is never evaluated if X is not a list.

APPLY **Subr**

```
(APPLY function argumentlist)
```

As usual for a Subr, APPLY evaluates both its arguments. The first must evaluate to a Subr atom or to an Expr list. The second must evaluate to a list of arguments for the function. The function is then applied to the arguments and the value of this is returned. The difference between this and a normal Subr or Expr application is that the arguments are not evaluated again before the function is applied. For example:

```
(APPLY CONS '(A B))
```

is equivalent to

```
(CONS 'A 'B)
```

and has the value

```
(A . B)
```

Another example is:

```
(APPLY '(LAMBDA (X Y) (SET X Y))
       '(A B))
```

The expression returns the value B, and A will be set to have the value B. The LAMBDA list must have a quote because the first argument of APPLY is always evaluated.
Note that

```
(APPLY SET '(A B))
```

would cause an error because SET is an Fsubr, not a Subr or Expr.

ATOM **Subr**

```
(ATOM argument)
```

ATOM returns T if its argument is any atom, character, number, Subr or Fsubr. If the argument is a list or dotted pair ATOM returns NIL. For example:

```
(ATOM 'CHARATOM)
```

has value T.

```
(ATOM CAR)
```

has the value T because CAR is a Subr atom.

```
(ATOM '(TAWNY SNOWY))
```

has value NIL.

BLANK **Character atom**

The initial value of BLANK is a single space character. It is particularly useful in separating items in PRINT functions.

CALL **Subr**

(CALL address accumulator)

CALL provides a method of using machine code subroutines from LISP. The first argument is numeric and supplies the address of the subroutine. If the address is larger than 32767 the equivalent two's-complement negative number must be used. The second argument defines the contents of the microprocessor's accumulator on entry to the routine and is optional. This argument must also be numeric. The low byte of the number is loaded into the accumulator. CALL returns a number giving the contents of the accumulator when the routine returns. For example:

(CALL -29)

calls a monitor routine which waits until a key is pressed and then puts the result in the accumulator.

CAR CDR **Subr**
CAAR CADR
CDAR CDDR

(CxxR list)

These functions return a part of the list obtained by evaluating their argument. It is an error for the argument to be an atom. CAR picks out the first member of a list or the first member of a dotted pair. CDR returns the remainder of the list with the first member removed. Applied to a dotted pair, CDR returns the second member. For example:

(CAR '(FORD LEYLAND DATSUN))

has value FORD.

(CDR '(FORD LEYLAND DATSUN))

has value (LEYLAND DATSUN).

(CAR '(DRIVE . LEFT))

has value DRIVE.
CAAR etc. provide for compound uses of CAR and CDR. For example:

(CADR X)

is equivalent to

(CAR (CDR X))

CHARACTER **Subr**

(CHARACTER number)

CHARACTER returns a single character. The ASCII code for the character is given by the number which is the value of the argument. This is useful for generating characters not available on the keyboard. For example:

(CHARACTER 65)

has the value A

(CHARACTER 7)

is the BELL character

CHARCOUNT **Expr**

(CHARCOUNT number item)

Returns the value of the first argument minus the number of characters required to print 'item', with numbers assumed to require 6 characters. CHARCOUNT is used by SPRINT. Its definition is as follows:

```
(LAMBDA
   (X LEFT)
   (COND
      ((ATOM X)
         (COND
            ((GREATERP
               LEFT (CHARS X)) (DIFFERENCE LEFT (CHARS X)))))
      (T
         (LOOP
            (UNTIL
               (ATOM X)
               (CHARCOUNT
                  X
                  (DIFFERENCE LEFT (COND (X 4) (T -2)))))
            (WHILE
               (SETQ
                  LEFT
                  (CHARCOUNT (CAR X) (DIFFERENCE LEFT 1))))
            (SETQ X (CDR X))))))
```

CHARP **Subr**

(CHARP expression)

CHARP returns T if its argument is a character atom. Otherwise it returns NIL. For example:

(CHARP A)

has value T

(CHARP 7)

has value NIL.

CHARS **Subr**

(CHARS characteratom)

Returns the number of characters in the character atom given. For example:

```
(CHARS 'HELLO)
```

has the value 5.

CLOSE **Subr**

(CLOSE <handle>)

Closes a current file and if necessary writes an EOF (end-of-file) marker. If the handle is 0, all current files are closed whatever their handles.

COND **Fsubr**

```
(COND (condition action action ...)
   (condition action action ...) ...)
```

COND is the main structure provided in LISP for testing and acting on conditions. COND can have any number of arguments. These arguments are treated in a special way.

The LISP interpreter looks at the arguments of COND one by one in the order in which they appear. Each argument is a list. The value of the first member of the list - the condition - determines the action taken. If the value is NIL, the rest of that list is ignored and LISP goes on to the next argument. If the value is T or anything other than NIL, the remaining members of the list are evaluated one by one. The value of the last member is returned as the value of the COND expression. There is no limit on the number of members of the list.

There are two special cases:

a) All of the conditions evaluate to NIL. In this case the value of COND is NIL.
b) An argument list has no actions, only a condition. Here the value of the condition is returned if it is not NIL.

For example:

```
(COND
   ((EQ X 3) (PRINT 'THREE) 'YES)
   (T 'NO))
```

This will print THREE and return the value YES if the value of X is 3. Otherwise it will return the value NO. COND expressions very often finish with a T condition to cope with all remaining possibilities.

Another example is:

```
(COND
    ((OR (MINUSP HOUR) (GREATERP HOUR 23))
        (ERROR '(HOUR NOT TIME OF DAY)))
    ((LESSP HOUR 7) 'NIGHT)
    ((LESSP HOUR 12) 'MORNING)
    ((LESSP HOUR 18) 'AFTERNOON)
    ((LESSP HOUR 22) 'EVENING)
```

```
    (T 'NIGHT))
```

This returns the time of day depending on the value of HOUR. There is an error message if HOUR does not represent a valid time.

CONS **Subr**

```
(CONS arg1 arg2)
```

CONS returns its two arguments combined as the dotted pair:;

```
(arg1 . arg2)
```

Usually arg2 will be a list, in which case this will have the effect of adding arg1 to the beginning of the list. CONS is the reverse of CAR and CDR. For example:

```
(CONS (CAR X) (CDR X))
```

has the same value as X:

```
(CONS 'HOVERCRAFT '(SHIP AEROPLANE))
```

has the value

```
(HOVERCRAFT SHIP AEROPLANE)
```

CONS can be used to create dotted pairs:

```
(CONS 'X 'Y)
```

has the value (X.Y).

CR **Character atom**

The value of CR is a character atom of one character.
The character is a return, hexadecimal $0D

DEFUN **Fexpr**

```
(DEFUN functionname parameterlist action action ... )
```

DEFUN is a convenient method of defining functions. None of the arguments are evaluated. This use of DEFUN is exactly equivalent to:

```
(SETQ functionname
    '(LAMBDA parameterlist action action....))
```

The value returned by DEFUN is the function name. 'functionname' is the name of the function you wish to define. 'parameterlist' is the list of arguments and local variables which the function uses (see Section 4.3.2). Any number of actions can be given for the function to carry out. For example:

We wish to define the function ADD2 which adds two to a number.

```
(DEFUN ADD2 (X) (PLUS X 2))
```

ADD2 would then have the value

```
(LAMBDA (X) (PLUS X 2))
```

and (ADD2 36)

would have the value 38 as required.

DIFFERENCE Subr

(DIFFERENCE number1 number2)

The value returned is number1 minus number2. For example, if A has the value 16 and B has the value 3 then :

(DIFFERENCE A B)

has value 13.

DOLLAR Character atom

The value of DOLLAR is the character atom $. It is useful to use DOLLAR rather than the dollar sign in programs to avoid confusing the READ routines which expect $ to introduce an escape sequence.

EDIT Expr

(EDIT expression)

Details of EDIT together with an example of its use are given in Chapter 6. Its definition is as follows:

```
(LAMBDA
   (A (Q))
   (LOOP
      (SETQ Q (PRIN0 (GETCHAR)))
      (UNTIL (EQ Q 'B) A)
      (SETQ
         A
         (COND
            ((EQ Q 'R) (PRINT) (READ))
            ((EQ Q CR) (PRIN0 A) (PRINT) A)
            ((EQ Q 'C) (PRINT) (CONS (READ) A))
            ((ATOM A) (PRIN0 (QUOTE *)) A)
            ((EQ Q 'D) (CONS (CAR A) (EDIT (CDR A))))
            ((EQ Q 'A) (CONS (EDIT (CAR A)) (CDR A)))
            ((EQ Q 'X) (CDR A))
            (T (PRIN0 '?) A)))))
```

EOF Subr

(EOF <handle>)

Detects whether an end-of-file marker has been reached when reading. It returns T if end-of-file has been reached and NIL if it has not.

EQ Subr

(EQ arg1 arg2)

EQ will return T if one of the following conditions applies to the value of the two arguments:

a) They are the same character atom.
b) They are equal numbers.
c) They are identical lists in LISP memory.

Otherwise EQ returns NIL. For example:

```
(EQ 4 4)
```

has value T.

```
(EQ 'FRED 'FRED)
```

has value T.

```
(EQ 'FRED 4)
```

has value NIL.

For a pair of lists to be equal they must share the same memory as well as being identical in form. Usually EQUAL (see Chapter 7) is more useful on lists.

ERROR **Subr**

```
(ERROR arg arg ...)
```

ERROR behaves like PRINT in that it prints a return followed by the value of each of its arguments. Having done that it generates error number 15 and the usual error traceback occurs. Here is an example of its use in checking that L is a list before finding its CDR.

```
(COND ((ATOM L) (ERROR L BLANK 'NOTLIST))
      (T (CDR L)))
```

ERRORSET **Fsubr**

```
(ERRORSET expression)
```

Normally when an error accurs in evaluating an expression, the traceback works through all the function calls and halts the program. ERRORSET is a means of preventing this and keeping control in the program. 'expression' is an s-expression in which you think that an error could occur. The possible outcomes are as follows:

a) If there is no error, ERRORSET acts like LIST, i.e.

   ```
   (ERRORSET expression)
   ```

   is equivalent to

   ```
   (LIST expression)
   ```

b) If an error occurs then the value is a numeric atom giving the error number.

ATOM or LISTP therefore provide an easy way of determining whether an error has occured. The following would be a typical use of ERRORSET:

```
(LOOP (SETQ X (ERRORSET (READ)))
      (UNTIL (LISTP X) (CAR X))
      (PRINT '(TRY TYPING THAT AGAIN)) )
```

EVAL **Subr**

(EVAL arg)

EVAL returns its argument evaluated one extra time. As EVAL is a Subr, the total effect is that arg is evaluated twice. For example:

```
(SETQ CRITERION 'COLOUR)
(SETQ COLOUR 'GREEN)
```

After this

(EVAL CRITERION)

has the value GREEN.

F **Special character atom**

The initial value of F is NIL. It is intended that F is used for the logical value 'false' just as T is used for 'true'.

FSUBRP **Subr**

(FSUBRP arg)

FSUBRP tests whether its argument is a Fsubr atom. If so it returns T, if not it returns NIL. For example:

(FSUBRP COND)

has value T.

(FSUBRP 'COND)

has value NIL because COND is a character atom, even though its value is an Fsubr atom.

GET **Subr**

(GET characteratom propertyname)

GET searches the property list of the character atom for the given property name. If the name is found, the property is returned. If there is no such property, the value of GET is NIL. For example:

If ALBERT has properties HEIGHT 172
EYES BLUE
CONVICTIONS NIL

(GET 'ALBERT 'HEIGHT)

has value 172.

(GET 'ALBERT 'HAIRCOLOUR)

has value NIL.

(GET 'ALBERT 'CONVICTIONS)

has value NIL.
Note that it is not possible to distinguish between an absent property and a NIL property using GET.

GREATERP **Subr**

(GREATERP number1 number2)

GREATERP returns T if number1 is greater than number2. Otherwise the value is NIL. For example:

(GREATERP 4 3)

has value T.

GETCHAR **Subr**

(GETCHAR)

GETCHAR returns a single-character character atom. This character is the next byte to be input via the current input channel. Normally GETCHAR will return the next key to be pressed on the ATOM keyboard.

LAMBDA **Special character atom**

(LAMBDA parameterdefinition action action ...)

LAMBDA is used as the first member of Expr and Fexpr function definitions. The full syntax of function definitions is discussed in Chapter 4. LAMBDA has the value LAMBDA.

LESSP **Subr**

(LESSP number1 number2)

LESSP returns T if number1 is numerically less than number2. Otherwise it returns NIL. Both arguments must be numeric. For example:

(LESSP 5 10)

has value T.

(LESSP 6 6)

has value NIL.

LINEWIDTH **Number**

LINEWIDTH is used by SPRINT as the approximate limit on the number of characters printed per line. For example:

(SETQ LINEWIDTH 31)

will set the correct line width for the screen. For use with a printer, values of 78 or 130 might be more appropriate.

LIST **Subr**

(LIST arg1 arg2 ...)

List can have up to 28 arguments. After evaluating them all, as normal for a Subr, it makes up a list with the values as the members. This list is returned. If there are no arguments the value is NIL. For example:

```
(LIST 'A 3 '(X  Y))
```

has value (A 3 (X Y)).

LISTP **Subr**

(LISTP arg)

LISTP returns T if the argument is a list or a dotted pair, and NIL if the argument is an atom. It is the opposite of ATOM in that:

```
(NULL (ATOM X))
```

is equivalent to

```
(LISTP X)
```

For example:

```
(LISTP (CONS 'A X))
```

has value T.

```
(LISTP 3)
```

has value NIL.

LOAD **Subr**

(LOAD filename)

This command is described in Chapter 2.
For example:

```
(LOAD 'BASE)
```

loads the disc file named BASE.

LOOP **Fsubr**

(LOOP arg1 arg2 ...)

LOOP can have any number of arguments. LOOP repeatedly evaluates all its arguments, starting each time with arg1 and working in order through the list. A LOOP will never terminate unless one of the arguments contains a WHILE or UNTIL function. When the WHILE or UNTIL function is satisfied the LOOP will halt just before it evaluates its next argument. The value returned is the value of the last argument evaluated, i.e. the one containing the satisfied

WHILE or UNTIL. The following will print

X IS n

for values of n from 6 to 0 and then return DONE:

```
(SETQ X 7)
(LOOP (UNTIL (MINUSP (SETQ X (SUB 1 X)))
             'DONE)
      (PRINT 'X BLANK 'IS BLANK X) )
```

LPAR **Character atom**

LPAR has the character atom '(' as its value.

MESSON **Subr**
MESSOFF

(MESSON messagenumber)
(MESSOFF messagenumber)

MESSON and MESSOFF are used to control whether certain system messages are printed. MESSON will allow the message to be printed and MESSOFF will suppress it. Once the status of a message has been set this way it remains unchanged until a disastrous error occurs or another MESSON or MESSOFF expression is evaluated. The message numbers are as follows:

Number	Message
1	Garbage collection bytes collected
2	Garbage collection number
4	Error number
8	Error top level ARG'S
16	Error traceback
128	Read depth prompt

The value returned is the message number. For example:

(MESSON 16)

will cause error tracebacks to be printed in future.

(MESSOFF 16)

will prevent error tracebacks from being printed. Note that message numbers can be added together, so for example:

(MESSOFF 3)

will turn off both garbabge collecter messages simultaneously.

MINUSP **Subr**

(MINUSP argument)

MINUSP returns the value T if its argument is a negative number atom. Otherwise the value is NIL. For example:

(MINUSP -4)

has value T.

(MINUSP 'FRED)

has value NIL.

NIL **Special atom**

NIL is a special character atom. It always has NIL as its value. It is used to terminate lists and as the false value in logical expressions.

NOT **Subr**
NULL

(NOT argument)
(NULL argument)

NOT and NULL behave in exactly the same way. They return the value T if the argument is NIL. Otherwise they return NIL. For example:

(NOT T)

has value NIL.

(NULL '(1 2 3))

has value NIL.

(NULL F)

has value T. The value of F is NIL.

NUMBERP **Subr**

(NUMBERP argument)

NUMBERP returns T if its argument is a number atom. Otherwise the value is NIL. For example:

(NUMBERP (PLUS 6 5))

has value T.

(NUMBERP 'LETTERS)

has value NIL.

OBLIST **Subr**

(OBLIST)

OBLIST returns a list of all the character atoms known to LISP, except those that:

a) have the value UNDEFINEDVALUE
b) and have no properties

These conditions eliminate those atoms which are being used as character strings rather than as atoms with an interesting value. The order of the atoms in the list is the order in which they

appear in the ATOM memory, highest addresses coming first.

OPEN **Subr**

(OPEN <filename> <mode>)

The OPEN function opens the named file for input or output. If mode is T this indicates that the file already exists. If mode is NIL then a new file will be created. OPEN returns the file handle.

OR **Fsubr**

(OR predicate predicate ...)

OR returns NIL only if the values of all its arguments are NIL. Otherwise it returns T. OR evaluates its arguments one by one from the left hand end until it finds one whose value is not NIL. OR then returns T, without evaluating the remaining arguments. If OR reaches the end of the argument list without finding a non-NIL value it returns NIL. For example:

(OR 'X P Q)

has value T and P and Q do not get evaluated.

(OR NIL NIL)

has value NIL.

(OR)

has value NIL.

ORDINAL **Subr**

(ORDINAL characteratom)

ORDINAL returns the numeric ASCII code for the first character in the character atom. If the character has zero length the number zero is returned. It is an error for the argument not to be a character atom. For example:

(ORDINAL 'ALPHABET)

has value 65.

PEEK **Subr**

(PEEK address)

PEEK returns a number which represents the contents of the memory location whose address is given in the argument. The address must be a number atom. Addresses above 32767 can be represented by their equivalent two's-complement negative number. For example:

(POKE 768 21)

will place 21 in location 768 (see POKE).

(PEEK 768)

will then have the value 21.

PERIOD **Character atom**

PERIOD is a character atom consisting of one full stop character.

PLIST **Subr**

(PLIST characteratom)

PLIST returns the property list of the character atom. If the atom has no properties it returns NIL. The property list is a list of dotted pairs of the form:

((propname . property) (propname . property) ...)

For example:

if JACQUELINE has no properties to begin with:

```
(PUT 'JACQUELINE 'AGE 22)
(PUT 'JACQUELINE 'FLOWER 'TULIP))
```

Then

(PLIST 'JACQUELINE)

has the value

((FLOWER . TULIP) (AGE . 22))

PLUS **Subr**

(PLUS number number ...)

PLUS returns the sum of all its arguments, which must all be number atoms. PLUS can have any number of arguments up to 28. For example:

(PLUS 6 2 -3)

has value 5.

(PLUS)

has value 0.

POKE **Subr**

(POKE address number)

POKE stores the single byte representation of its second argument in the memory location specified by the first argument. Both arguments must be numbers. For example:

(POKE 34 TOP)

PRIN0 **Subr**

(PRIN0 arg1 arg2 ...)

PRIN0 prints its arguments one by one, starting from the current

printing position. Because PRIN0 is a Subr, all the arguments are evaluated before printing starts.
N.B. No spaces or other separators will be placed between the arguments - this must be done explicitly using BLANK or CR. PRIN0 returns the value of its last argument. For example:

(PRIN0 'PRICE BLANK DOLLAR 32)

would print

PRICE $32

and has value 32.

PRINT **Subr**

(PRINT argl arg2 ...)

PRINT is exactly like PRIN0 except that printing begins on a new line.

PROGN **Fsubr**

(PROGN expression expresion ...)

PROGN evaluates its arguments one by one and returns the value of its last argument. It is useful in allowing several expressions to be evaluated where the syntax would otherwise only allow for one. For example:

if X has the value 5

(PROGN (PRINT X DOLLAR) (LIST X))

would print 5$ and have the value (5).

PUT **Subr**

(PUT characteratom propertyname property)

PUT places the property on the property list of the character atom referenced by the property name. The value of the PUT expression is the property, the third argument. An atom never has two properties with the same name: a second PUT with the same property name will replace the old property. PUT is well illustrated by this function to save the present value of an atom on its property list as the OLDVALUE property:

(LAMBDA (X)

(PUT X 'OLDVALUE (EVAL X)))

The value could then be restored by;

(SETQ GULL (GET 'GULL 'OLDVALUE))

QUOTE **Fsubr**

(QUOTE argument)

QUOTE returns its argument unevaluated. It is equivalent in effect to the single quote mark. For example:

(QUOTE APPLES)

has value APPLES.

'PEARS

has value PEARS.

QUOTIENT **Subr**

(QUOTIENT dividend divisor)

QUOTIENT returns the integer part which results from dividing the dividend by the divisor. Both arguments must be numbers. For example:

(QUOTIENT 5 2)

has value 2.

(QUOTIENT -400 6)

has value -66.

READ **Subr**

(READ)

READ reads one s-expression from the current input stream (e.g. the keyboard or disc file) and returns it unevaluated. No prompt is given.

READLINE **Subr**

(READLINE <handle>)

Returns all the characters upto the next carriage return from the file whose handle is given as a single character atom.

RECLAIM **Subr**

(RECLAIM)

RECLAIM causes a garbage collection. The value returned is always NIL. With the garbage collection messages on, this is a useful way of finding out how much free memory is left.

REMAINDER **Subr**

(REMAINDER dividend divisor)

REMAINDER returns the remainder from the division of the dividend by the divisor. This value is always positive. Both arguments must be numbers. For example:

(REMAINDER 5 2)

has value 1.

(REMAINDER -400 66)

has value 4.

REMPROP **Subr**

(REMPROP characteratom propertyname)

REMPROP will remove a property from the property list of an atom. The value is T if there was originally such a property on the list and NIL if there was not. For example, after

(PUT 'EAGLE 'ISBIRD T)

the value of

(REMPROP 'EAGLE 'ISBIRD)

is T the first time that it is evaluated and NIL afterwards.

RPAR **Character atom**

RPAR is a single character atom consisting of a close bracket.

RPLACA **Subr**

(RPLACA list arg2)

RPLACA returns the list with its CAR replaced by the second argument. The first argument must be a list or dotted pair. There are no restrictions on the second argument. RPLACA should be used with caution as it actually alters the list cell in memory. Therefore other data structures using the same cell will also be changed. For example:

(RPLACA '(A B C) 'FIRST)

has value

(FIRST B C)

If the above use of RPLACA was part of a program, execution would alter the actual expression to

(RPLACA '(FIRST B C) 'FIRST)

RPLACD **Subr**

(RPLACD list arg2)

RPLACD returns the list with its CDR replaced by the second argument. It is therefore similar to RPLACA and the same cautions apply. For example:

(RPLACD '(A B C) 'FIRST)

has value (A . FIRST)

SAVE **Subr**

This function is described in Section 2.2. For example:

(SAVE 'BASE)

saves the workspace as the disc file BASE.

SET **Fsubr**

(SET characteratom newvalue)

SET evaluates both its arguments. The first must evaluate to a character atom. The value of this character atom is set to be the value of the second argument. This value is returned. For example:

(SET (QUOTE X) (PLUS 2 3))

has value 5 and X now has the value 5.

SETQ **Fsubr**

(SETQ characteratom newvalue)

SETQ evaluates only its second argument. The first argument must be a character atom. The value of the second argument becomes the value of this character atom, and is the value returned. For example:

(SETQ X (PLUS 25 -20))

has value 5 and X now has the value 5

SUBRP **Subr**

(SUBRP arg)

SUBRP tests whether its argument is a Subr atom. If so it returns T, otherwise NIL. For example:

(SUBRP SUBRP)

has value T.

(SUBRP COND)

has value NIL.

SPRINT **Expr**

(SPRINT arg columnindent)

SPRINT prints its first argument with the list structure neatly displayed. The column indent is optional. If supplied it must be zero or a positive number. Printing is offset by the column indent. SPRINT uses XTAB and CHARCOUNT as subroutines.

```
(LAMBDA
   (X (N . 0))
   (COND
      ((OR (ATOM X) (CHARCOUNT X (DIFFERENCE LINEWIDTH N)))
         (PRIN0 X))
      (T
         (PRIN0 LPAR)
         (SPRINT (CAR X) N)
         (SETQ N (PLUS N 3))
         (LOOP
            (SETQ X (CDR X))
            (COND ((AND X (ATOM X)) (PRIN0 PERIOD X)))
            (UNTIL (ATOM X) (PRIN0 RPAR))
```

```
        (XTAB N)
        (SPRINT (CAR X) N)))))
```

T **Character atom**

T has the value T. It is used as the 'true' logical value. For example:

```
(NUMBERP 43)
```

has value T.

TIMES **Subr**

(TIMES number number ...)

TIMES returns the product of all its arguments. The arguments must all be numeric. There can be any number of arguments up to 28. For example:

```
(TIMES 28 4 -2)
```

has value -224.

UNDEFINED **Special character atom**

UNDEFINED has the value UNDEFINED. When a new character atom is created by READ, GETCHAR or CHARACTER it is given this initial value. Atoms with the value UNDEFINED and no properties are special in that:

a) they do not appear on the OBLIST
b) they can be removed by the garbage collector if they are not accessed by any data structure

UNTIL **Fsubr**

(UNTIL predicate action action....)

UNTIL is used in conjunction with LOOP q.v. The predicate is evaluated. If its value is NIL, UNTIL returns NIL. If the value is not NIL, the following changes occur:

a) the actions are evaluated one by one and the last one is returned as the value of UNTIL. If there are no actions the value of the predicate is returned.
b) a flag is set to terminate the first surrounding LOOP. This termination does not occur until LOOP attempts to evaluate its next top level argument.

See LOOP for an example.

WHILE **Fsubr**

(WHILE predicate action action ...)

WHILE is used in conjunction with LOOP q.v. It is similar to UNTIL, except that the predicate must be NIL for the actions to be evaluated and the LOOP to terminate. If WHILE has no actions when the predicate is NIL, NIL is returned.

WRITE **Subr**

(WRITE handle item1 item2 ...)

Writes the given items to the file whose handle is given, preceeded by a carriage return.

WRITE0 **Subr**

(WRITE0 handle item1 item2 ...)

Writes the given items to the file whose handle is given.

XTAB **Expr**

(XTAB number)

Prints the specified number of spaces at the start of a new line. XTAB is used by SPRINT. Its definition is as follows:

```
(LAMBDA
   (S)
   (PRINT)
   (LOOP
      (UNTIL (MINUSP (SETQ S (DIFFERENCE S 1))))
      (PRIN0 BLANK)))
```

ZEROP **Subr**

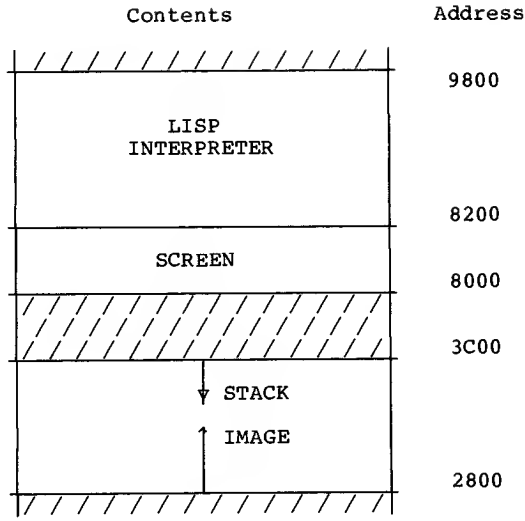
(ZEROP arg)

ZEROP returns T if the argument is the number zero. Otherwise it returns NIL. For example:

```
(ZEROP (REMAINDER 8 2))
```

has value T.

Appendix
Memory Map



FIRST EDITION

ISBN 0 907876 01 3